Exploiting Hierarchy in the Abstraction-Based Verification of Statecharts Using SMT Solvers

Bence Czipó[1] Ákos Hajdu[1,2] Tamás Tóth[1*] István Majzik[1]

[1]Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary

[2]MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

czipobence@gmail.com, {hajdua,totht,majzik}@mit.bme.hu

Statecharts are frequently used as a modeling formalism in the design of state-based systems. Formal verification techniques are also often applied to prove certain properties about the behavior of the system. One of the most efficient techniques for formal verification is Counterexample-Guided Abstraction Refinement (CEGAR), which reduces the complexity of systems by automatically building and refining abstractions. In our paper we present a novel adaptation of the CEGAR approach to hierarchical statechart models. First we introduce an encoding of the statechart to logical formulas that preserves information about the state hierarchy. Based on this encoding we propose abstraction and refinement techniques that utilize the hierarchical structure of statecharts and also handle variables in the model. The encoding allows us to use SMT solvers for the systematic exploration and verification of the abstract model, including also bounded model checking. We demonstrate the applicability and efficiency of our abstraction techniques with measurements on an industry-motivated example.

1 Introduction

Statecharts are frequently used for modeling and designing state-based systems. Such systems also appear in safety critical domains, thus ensuring their correct operation is gaining increasing importance. Formal verification techniques (such as model checking) can yield mathematically precise proofs regarding the correctness of a model of the system. A widely used requirement is safety, where the purpose of verification is to check if a given erroneous state configuration is reachable during the operation of a system. However, a typical drawback of using formal verification techniques is their high computational complexity, as the set of possible configurations for a system can be unmanageably large or even infinite. A possible solution to overcome this issue is to use abstraction, which is a generic technique for reducing complexity by hiding details that are not relevant for the property to be verified. However, it is a difficult task to find the proper precision of abstraction, which shall be coarse enough to avoid complexity issues but fine enough to prove the desired property. Counterexample-Guided Abstraction Refinement (CEGAR) is an automatic technique that initially starts with a coarse abstraction and refines it iteratively based on the counterexamples until the proper precision is obtained [8]. CEGAR was first described for transition systems [8] but since then it has been applied in various fields of verification [3, 11, 13].

In our paper we present a novel adaptation of the CEGAR approach to the reachability analysis of hierarchical statecharts. We first define an encoding of the statechart to logical formulas that preserves information about the hierarchy, such as parallel regions and composite states. Our approach also supports some additional elements of statecharts, including variables, events, guards and actions. Based on this encoding we propose an abstraction over the hierarchical structure by only expanding composite states

*Partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

Submitted to:
FESCA 2017

until a given depth. Refinement is performed by increasing the depth for certain states along a spurious counterexample. Furthermore, we also combine our state-based abstraction technique with the variable abstraction of Clarke et al. [11] to efficiently handle variables in the abstract statechart. The main novelty of our approach is that the encoding allows us to use SMT solvers [6] for the systematic exploration and bounded model checking of the abstract state space. We evaluate and demonstrate the applicability and scalability of our algorithms by performing reachability queries on an industry-motivated example.

The rest of this section discusses related work. Section 2 introduces preliminaries of our work. Section 3 presents our encoding of statecharts to logical formulas. Section 4 describes the adaptation of CEGAR to statecharts. Section 5 evaluates the algorithms and Section 6 concludes our work.

Related work. Several works in the literature address the formal verification of statecharts. Based on a survey [4] most approaches flatten the hierarchy of the statechart or transform the problem to the input language of a model checker such as SMV [7] or SPIN [15]. The disadvantage of these approaches is that the information in the state hierarchy is not preserved and it is often difficult to interpret the results on the original statechart. Alur et al. [1] exploit hierarchy, but they work with hierarchical reactive modules, where hierarchy has a bit different semantics than in statecharts: submodules can interact through interfaces and concurrency is only allowed at the top level.

The work of Meller et al. [19, 18] is the most related to our current paper. They also defined a CEGAR-like approach for statecharts, supporting a wide range of their elements. They focus on LTL_x model checking, while our approach currently only targets reachability. They use a model-to-model transformation, which means that the abstraction of a statechart is also a statechart similarly to our approach. They abstract a composite state using its interface, whereas we treat abstracted composite states as a simple state. The main difference however, is that in their approach the abstract model is transformed to the input language of a model checker, whereas in our method we encode the abstract statechart as SMT formulas, allowing us to use SMT solvers to perform CEGAR and to utilize the power of SMT-based model checking.

The recent work of Helke and Kammüller [14] also defines abstractions over statecharts for the universal fragment of CTL model checking, which is more general than our reachability analysis. However, their main focus is on only abstracting the data and preserving the structure of the statechart, whereas in our approach abstraction on the structure is also a key feature.

2 Background

In this section we first present hierarchical statecharts as the formalism used in our work (Section 2.1). Then, we describe model checking (Section 2.2) and we introduce Counterexample-Guided Abstraction Refinement (Section 2.3), an efficient model checking technique.

2.1 Hierarchical Statecharts

In our work we describe state-based event-driven behavior of systems using *hierarchical statecharts* [18]. Expressions and variables of the statechart are based on first order logic (FOL) [6]. Let FOL denote the set of all first order logic formulas. Let the formula $\psi \in \text{FOL}$ be a first order formula and $V = \{v_1, v_2, \ldots, v_k\}$ be the set of the variables appearing in ψ. Let V_i represent the indexed version of the variables, i.e., $V_i = \{v_{1,i}, v_{2,i}, \ldots, v_{k,i}\}$, and let ψ_i denote the formula, where each variable $v_j \in V$ is replaced by $v_{j,i}$ from V_i. For example if $\psi = v_1 \wedge v_2$ then $\psi_4 = v_{1,4} \wedge v_{2,4}$.

Hierarchical statechart. Formally, a hierarchical statechart [18] is a tuple $Sc = (S, R, par, I, V, Tr)$ where

- S is the finite set of *states* in the statechart,
- R is a finite set of *regions* in the statechart,
- $par : (S \cup R) \mapsto (S \cup R \cup \{root\})$ is the *hierarchy function* that assigns a parent (container) region to every state, and a parent (container) state or a distinguished *root* element of the statechart to every region in the statechart,
- $I : R \mapsto S$ assigns an *initial state* to each region in the statechart,
- V is the set of *variables* appearing in guards and actions,
- $Tr \subseteq S \times S \times EV \times G \times Act$ is the set of *transitions*, where EV is the set of *events*, $G \subseteq \mathsf{FOL}$ is the set of *guard* expressions and Act is the set of *actions*.

For a transition $t = (s, s', e, g, a) \in Tr$, let $src(t) = s$ and $trgt(t) = s'$ denote its *source* and *target* states, let $trig(t) = e$ denote its *trigger* event, let $grd(t) = g$ denote its guard expression, and let $act(t) = a$ denote the action executed when t fires. In our current work an action is either an event raising $raise(e)$ (where $e \in EV$) or a variable assignment $v := \psi$ (where $\psi \in \mathsf{FOL}$).

An example statechart can be seen in Figure 1. In their visual representation, states are marked with rectangles and regions are marked with dashed lines, but only if there are multiple regions contained in a state, for example in case of composite state A. Initial states are denoted by black dots and transitions are represented by arrows. The initial values of variables are described in a dashed block on the left of the statechart ($x := 0$). The transition from state B to A presents an example for guard ($x > 5$) whereas the transition from state $B2c$ to $B1$ has an assignment action ($x := x + 1$).

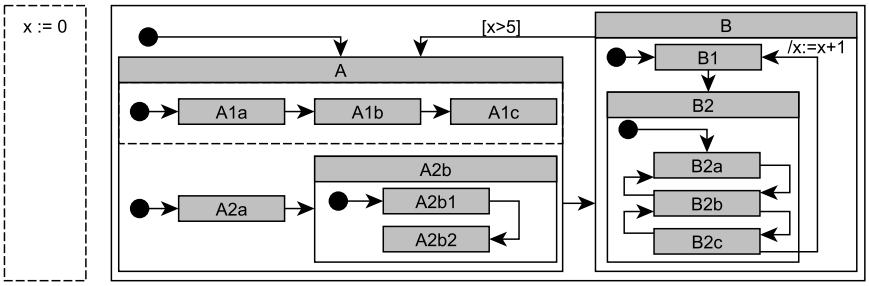


Figure 1: Example statechart with parallel regions and hierarchy.

Let *chld* denote the opposite direction of the hierarchy, i.e., $chld(x) = \{y \mid par(y) = x\}$. The set of *ancestor states* of a state $s \in S$ is $anc(s) = \{par(par(s))\} \cup anc(par(par(s)))$, i.e., its parent state and the ancestors of its parent. The set of *descendant states* for a state s is $desc(s) = \{s' \mid s' \in S, anc(s') = s\}$, i.e., states s' for which s is an ancestor. A state s is *simple* if $|desc(s)| = 0$ and *composite* otherwise.

Ancestors and descendants can be defined for regions as well. The state s is an ancestor to region r ($s \in anc(r)$) if $s = par(r)$ or $s \in anc(par(r))$, and the state s is the descendant of region r ($s \in desc(r)$), if there exists $s' \in chld(r)$ such that $s' = s$ or $s' \in anc(s)$.

The depth function $depth : S \mapsto \mathbb{N}$ assigns the number of its ancestor states (including the root object) to a state. Inductively defined, $depth(root) = 0$, and for every $s \in S$, $depth(s) = depth(par(par(s))) + 1$. The integer $d = \max(\{depth(s) \mid s \in S\})$ is the *maximum depth* of the statechart. Let the i-th *level* of the

hierarchy refer to the set of states with depth i, i.e., $\{s \in S \mid depth(s) = i\}$. In case of the statechart in Figure 1, the depth of state A is 1, $A1a$ is 2 and $A2b2$ is 3. The states of level 1 are $\{A,B\}$, level 2 are $\{A1a,A1b,A1c,A2a,A2b,B1,B2\}$ and level 3 are $\{A2b1,A2b2,B2a,B2b,B2c\}$.

Configuration. A *configuration* of a statechart is a tuple $c = (\omega, \rho, \alpha)$ where

- $\omega \subseteq S$ is a set of active states such that each top level region contains exactly one active state and child regions of an active state must also contain exactly one active state,
- $\rho \subseteq EV$ is the set of currently active events on the input of the statechart,
- α is the assignment to the variables V.

Let c_I denote the initial configuration of a statechart (determined by I), and let $C_{Sc} = \{c_1, c_2, \ldots\}$ denote all possible configurations of Sc. Note, that although S and R is finite, unbounded variables in V can make C_{Sc} infinite. Considering the example presented in Figure 1, an example configuration is $\{\{A,A1a,A2a\},\emptyset,\{x = 0\}\}$, which is also the initial configuration.

Transition Relation. The *transition relation* of a statechart Sc is a set $N \subseteq C_{Sc} \times C_{Sc}$. Given two configurations $c_1 = (\omega_1, \rho_1, \alpha_1)$ and $c_2 = (\omega_2, \rho_2, \alpha_2)$, $(c_1, c_2) \in N$ if a transition $t \in Tr$ exists for which the following conditions hold.

- The source state $src(t)$ of t is active in c_1 ($src(t) \in \omega_1$), its guard $grd(t)$ evaluates to true under α_1 and its trigger event is present on the input ($trig(t) \in \rho_1$),
- After taking t the set of actives states ω_2 is obtained by removing its source $src(t)$ its ancestors $anc(src(t))$ and its descendants $desc(src(t))$ from ω_1 and adding its target $trgt(t)$ and its ancestors $anc(trgt(t))$. If $trgt(t)$ is a composite state then the initial states of each of its regions are also added to ω_2 recursively. The set of active events ρ_2 is obtained by removing its trigger event $trig(t)$ from ρ_1 and adding the event $e \in EV$ if its action $act(t)$ is an event raising $raise(e)$. The assignment α_2 maps each variable to the same value as α_1, except the variable v if the action $act(t)$ is an assignment of the form $v := \psi$. In this case v is mapped to ψ in α_2.

Furthermore, for a configuration $c \in C_{Sc}$ let $N(c) = \{c' \in C_{Sc} \mid (c, c') \in N\}$ denote its *successors*, i.e., the set of the configurations that are reachable from c with a single transition. In our work transitions do not have priority, therefore successors are selected non-deterministically.

Path. A sequence of configurations $\pi = (c_0, c_1, \ldots, c_n)$ is a *path* in Sc if $c_i \in C_{Sc}$ (for $0 \leq i \leq n$), $(c_i, c_{i+1}) \in N$ (for $0 \leq i < n$) and $c_0 = c_I$. The length of a path is the number of transitions occurring in the path, so the length of a path with $n+1$ configurations is n. For the example statechart presented in Figure 1 a possible path is $\pi = (\{\{A,A1a,A2a\},\emptyset,\{x = 0\}\},\{\{A,A1a,A2b,A2b1\},\emptyset,\{x = 0\}\},\{\{B,B1\},\emptyset,\{x = 0\}\})$, with a length of 2. A configuration c_r is *reachable* if there is a path $\pi = (c_0, c_1, \ldots, c_n)$ leading to c_r, i.e, $c_n = c_r$. Let the set of all reachable configurations be denoted by $C_R \subseteq C_{Sc}$.

Bit vectors. Bit vectors are sequences of 0 (false) and 1 (true) bits. The set of bit vectors of length n is denoted by BV_n. The ith bit in the vector $bv \in BV_n$ is denoted by $bv[i]$. Bit vectors can be extended with *don't care* bits (X). Two bits are *conflicting* if they are not the same and none of them is X. The combination of non-conflicting bits b_1 and b_2 is b_1 if $b_1 = b_2$ or $b_2 = X$ and b_2 otherwise. Two bit vectors are conflicting if they contain conflicting bit pairs at any position. If two bit vectors (of the same length) are non-conflicting, they are *combinable*, and their combination is their bitwise combination.

2.2 Model Checking

Model checking [10] is a technique to verify systems against given requirements by systematically traversing the state space of the system. In our paper we focus on *safety requirements*. A statechart *Sc* is *safe* for a predicate $p\colon C_{Sc} \mapsto \{\top, \bot\}$ over C_{Sc} if for every reachable configuration $c_r \in C_R$, $p(c_r) = \top$ holds. If the statechart is not safe, a *counterexample* can be found, which is a path $\pi = (c_0, c_1, \ldots, c_n)$ with $p(c_n) = \bot$. Reachability and safety are opposites: a system is safe if no "bad" configuration is reachable.

State Space Exploration. The most basic way of checking a safety requirement is to systematically enumerate the set of reachable configurations C_R and to check if the predicate p holds. This can be done by first starting from the initial configuration ($C_{R0} = \{c_I\}$) and then iteratively adding configurations that are reachable in one step from the already reached configurations ($C_{Ri+1} = C_{Ri} \cup \{c' \mid (c, c') \in N, c \in C_{Ri}\}$) until a fixpoint is reached ($C_{Ri+1} = C_{Ri}$). However, as C_R can be large or even infinite, this method is only applicable for large systems with additional techniques, for example CEGAR (Section 2.3).

Bounded Model Checking. *Bounded Model Checking* (BMC) [5] is an iterative algorithm to check if a safety requirement holds within a given bound k. A configuration c_k for a statechart *Sc* is considered *k-reachable* if there is a path $\pi = (c_0, c_1, \ldots, c_k)$ in *Sc* leading to c_k with length k. Bounded model checking iteratively checks the safety of k-reachable configurations, incrementing k from 0 to an upper bound (or until a counterexample is found).

Bounded model checking is realized by transforming k-reachability to a SAT or SMT formula [5] such that c_k is reachable if and only if the formula is satisfiable. In case of a *transition system* that only contains states (S), transitions ($T \subseteq S \times S$) and initial states ($I \subseteq S$) the widely used approach is to assign a unique bit vector to each state $s \in S$. Such vectors are then transformed into formulas by assigning a boolean variable to each bit and forming a conjunction of them in the following way: if a bit is 0 then its corresponding variable is negated. Let the formula assigned to state $s \in S$ be denoted by $form(s)$. Then, a transition $(s, s') \in T$ at step i is expressed as $form((s, s'))_i = form(s)_i \wedge form(s')_{i+1}$, and the whole transition relation is expressed as $form(T)_i = \bigvee_{t \in T} form(t)_i$, i.e., the disjunction of formulas assigned to transitions. Reachability in k steps can be decided by solving the formula $(\bigvee_{s \in I} form(s)_0) \wedge (\bigwedge_{i=0}^{k} form(T)_i)$, which is also called as *unfolding* the transition relation k times. Furthermore, a satisfying assignment to the variables also determines the bit vectors, thus states along the path can be reconstructed. Consequently, this technique can also be used in state space exploration when enumerating states reachable in one step. However, this technique cannot be applied directly to hierarchical statecharts, as their configurations can contain multiple active states and they also have additional constructs like guards and actions. In the next section we propose a generalization of this transformation approach to support hierarchical statecharts and their elements.

2.3 CEGAR

Formal verification methods face difficulties handling large and sophisticated systems as their set of reachable configurations can be large or even infinite. *Abstraction* is a general mathematical approach to simplify the model checking problem by hiding irrelevant information from the system. In this paper we restrict to *existential* abstractions, that are over-approximating the original system, i.e., they might introduce additional behavior. A major issue with abstraction-based methods is to find the proper precision of abstraction that is fine enough to prove the desired requirement but coarse enough to reduce complexity.

Counterexample-Guided Abstraction Refinement (CEGAR) [8] is a general algorithm to automatically find the required precision of the abstraction by refining it based on counterexamples. The algorithm was first described for transition systems [8] but since then it has been applied in various fields [3, 11, 13].

CEGAR-based algorithms usually consist of four main steps. The first step is to create an *initial abstraction* from the original system. Then the abstract system is *checked* against the requirement (for example using state space exploration or BMC). If the requirement holds, due to the existential property of the abstraction, it also holds for the original system. Otherwise, an abstract counterexample exists, that has to be checked whether it is feasible in the original system (*concretization*). If it succeeds, a counterexample is found in the original system (witnessing that the requirement does not hold). Otherwise, the counterexample is only caused by a behavior introduced by the abstraction, so it is called *spurious* and the abstraction has to be *refined*. After the refinement, the abstract system can be checked again and this process is repeated.

3 Hierarchy Preserving Encoding of Statecharts

In this section we present a novel technique to transform hierarchical statecharts to logical formulas. We first describe an encoding that assigns bit vectors to states exploiting the hierarchy (Section 3.1) and then we extend the transformation of bit vectors to logical formulas supporting the additional constructs of statecharts (Section 3.2).

3.1 Encoding States to Bit Vectors

In order to assign bit vectors to the states of a statechart using the hierarchy, two main problems have to be addressed. First Section 3.1.1 presents an encoding for statecharts containing only parallel regions (but no hierarchy) and then Section 3.1.2 generalizes this encoding for any kind of hierarchical statechart.

3.1.1 Encoding Parallel Regions

Our main idea of encoding states in parallel regions is that each region gets a fixed segment in an n bit long bit vector with each segment being encoded independently. This way we can refer to a single state by omitting the other segments (filling their bits with don't care values) and we can also refer to a configuration by joining the segments of the active states in each region.

Formally, for a region r let $bits(r)$ denote the minimum number of bits required to assign each state in r a unique bit vector, i.e., the length of its associated segment. If there is no hierarchy (only parallelism) then $bits(r) = \lceil \log_2(|chld(r)|) \rceil$. Furthermore, given a set of regions $R = \{r_1, r_2, \dots r_k\}$ let $offs(r_i)$ denote the offset (starting position) of the segment of each region. The offset can easily by calculated for the ith region by summing the size of the preceding segments: $offs(r_i) = \sum_{j=1}^{i-1} bits(r_j)$.

Then for a non-hierarchical statechart with parallel regions $R = \{r_1, r_2, \dots, r_k\}$ let $enc\colon S \mapsto BV_n$ assign a bit vector of length $n = \sum_{j=1}^{k} bits(r_j)$ to each state in the following way.

1. First for each region r_i let enc_i assign a locally unique bit vector $enc_i(s)$ of length $bits(r_i)$ to each state $s \in chld(r_i)$ of the region. This can easily be done for example by simply numbering the states from 0 to $|chld(r_i)| - 1$ and encoding them into binary form.
2. Then for each state $s \in S$ with $s \in chld(r_i)$ let the assigned bit vector $enc(s)$ be defined in the following way: $enc(s)[j] = enc_i(s)[j - offs(r_i)]$ if $offs(r_i) < j \leq offs(r_i) + bits(r_i)$, and X otherwise.

In other words, the associated segment is filled with the locally unique bits while other segments are filled with don't care bits.

The advantage of our encoding is that transitions within a region can be translated to a logical formula using the encoding of their source and target states, without affecting the other regions. Furthermore, the set of active states in a configuration can also be encoded by taking the combination of the bit vectors corresponding to each state.

3.1.2 Encoding Hierarchy

Our main idea of encoding hierarchically nested states is to express containment in bit vectors. Similarly to parallel regions, we do this by assigning each level of the hierarchy a fixed segment in the bit vector with each segment being encoded independently. This way we can refer to a (possibly composite) state on the ith level by omitting the segments after the ith index (filling their bits with don't care values), meaning that we do not care or know about states below the ith level.

Formally, let $bits(i)$ denote the minimum number of bits required to encode the ith level, which can be calculated in the following way. For a region $r \in R$, let $bits(r)$ be the number of minimum bits required to encode the region, assuming that each contained state is simple, i.e., $bits(r) = \lceil \log_2(|chld(r)|) \rceil$ just as before. For a composite state and the root object $s_c \in S \cup \{root\}$, let $bits(s_c)$ be $\sum_{r \in chld(s_c)} bits(r)$, i.e., the sum of the minimum bits required to encode each child region and for a simple state $s_s \in S$, let $bits(s_s)$ be 0 as it contains no regions.

Different states on the same hierarchy level may contain different number of descendant states, therefore requiring different number of bits for encoding. We want to be able to refer to any of the states, therefore $bits(i) = \max(\{bits(s) \mid s \in S, depth(s) = i\})$. In other words, for the ith level we have to take the maximum of the minimum number of bits to encode each state under that level. Note, that in the deepest level, there is no composite state (otherwise there would be another level), so $bits(d) = 0$.

As for parallel regions, an offset $offs(i)$ can be defined for each hierarchy level that determines the starting position of its encoding in the bit vector. Again, the offset can be calculated for the ith level by summing the size of the preceding levels: $offs(i) = \sum_{j=0}^{i-1} bits(j)$. Then states can be encoded to a bit vector of length $n = \sum_{i=0}^{d} bits(i)$ in the following way.

1. First for each level $0 \leq i \leq d$ let enc_i assign a locally unique bit vector $enc_i(s)$ of length $bits(i)$ to each state s on the ith level ($depth(s) = i$) as if they were simple states. If there are no parallel regions, this can be done by numbering, otherwise the encoding of parallel regions (presented in Section 3.1.1) can be applied. By convention, we always give the number 0 to the initial states. We will rely on this convention later when transforming transitions.
2. Then for each state $s_i \in S$ with $depth(s) = i$ and ancestors $\{s_1, s_2, \ldots, s_{i-1}\}$ let the assigned bit vector $enc(s)$ be defined as the concatenation of $enc_j(s_j)$ for the first $offs(i) + bits(i)$ bits ($0 \leq j < i$) and let the remaining bits be filled with X bits.

The advantage of our encoding is that if the source or target of the transition is a composite state, it implicitly implies that any descendant state can also take the transition. Furthermore, the set of active states in a configuration can be encoded by taking the combination of the bit vectors corresponding to each state, since a child state has the same prefix as its parent.

As an example, consider the statechart in Figure 1. A possible encoding for this statechart is presented in Figure 2. For the ease of understanding, segments corresponding to levels are separated by dots. On the first level, there are two states A and B which requires a single bit. As A is the initial state, it is

assigned 0 and B is assigned 1. The second level is more complicated. B only contains $B1$ and $B2$ (requiring a single bit), but A has two regions containing 3 and 2 states respectively, which requires $2+1=3$ bits. Therefore, the second level is encoded in $\max(1,3)=3$ bits. As $B1$ is the initial state, it gets $XX0$ and $B2$ gets $XX1$. The reason behind don't care bits here is that we only have 2 states out of the 8 that could be encoded with 3 bits. It is just a convention, replacing these X bits with 0 bits would not make any difference. Encoding states in A require the rule for parallel regions. In the 3 bit long segment of the second level, the first two bits are used for the top region ($A1a, A1b, A1c$) and the third bit is used for the bottom region ($A2a, A2b$). On the third level there are 2 states in $A2b$ and 3 states in $B2$, requiring $\max(1,2)=2$ bits for local encoding.

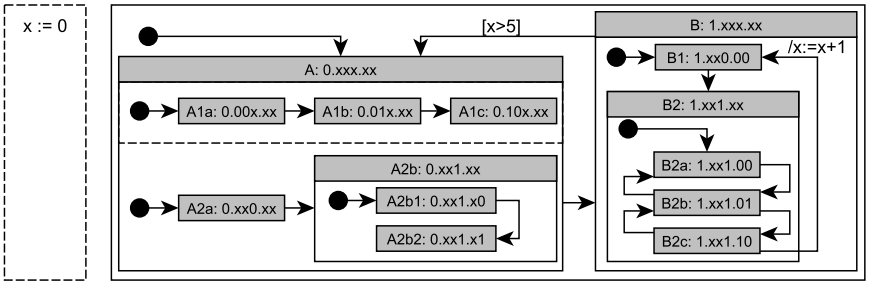


Figure 2: Encoding of an example statechart.

As the example shows, bit vectors assigned to active states in a configuration can be combined, for example for the active states $\{A, A1c, A2b, A2b2\}$ the bit vectors assigned to the states are $0.XXX.XX$, $0.10X.XX$, $0.XX1.XX$, $0.XX1.X1$ respectively so their combination is $0.101.X1$.

3.2 Transformation to Logical Formulas

Based on the encoding function *enc* defined in the previous section we now describe the transformation of statecharts to logical formulas. Let $f\colon BV_n \mapsto \mathsf{FOL}$ be a function that assigns formulas to bit vectors over the set of variables $V=\{v_1, v_2, \ldots v_n\}$ (distinct from the variables appearing in the statechart) in the following way: $f(bv)=\bigwedge_{i=1}^{n} lit(bv[i], i)$ where $lit(b,i)$ is $\neg v_i$ if $b=0$, v_i if $b=1$, and $\top$ if $b=X$. For example, the formula assigned to the bit vector $01X0$ is $\neg v_1 \wedge v_2 \wedge \top \wedge \neg v_4$. Note, that due to the semantics of the X bit, this formula has two satisfying assignments corresponding to 0100 and 0110.

Using the functions *enc* and f, the function *form* defined in Section 2.2 can be extended to hierarchical statecharts. For a state s let $form(s)$ be $f(enc(s))$, and for a set of active states ω let $form(\omega)$ be $f(enc(\omega))$.

To assign formulas to transitions, not only source and target states, but also trigger events, guards and actions need a subformula to be assigned. As there are finite number of events, they can also be assigned bit vectors and formulas $form(e)$ ranging over a set of (fresh) variables. Guards are FOL formulas, so in case of a guard $g \in G$, $form(g)=g$. In our current work we restrict the set of actions to variable assignments and event raising. An assignment $v_j := \psi$ in the ith step can be expressed with the formula $v_{j,i+1}=\psi_i$, whereas raising event $raise(e)$ can be expressed as $form(e)_{i+1}$. For a transition t to fire at step i, each of these formulas have to be satisfied, so

$$form(t)_i = form(src(t))_i \wedge form(trgt(t))_{i+1} \wedge form(trig(t))_i \wedge grd(t)_i \wedge form(act(t))_i.$$

The formula above works well for transitions with a composite source: any descendant state will be able to take the transition. However, it is not suitable if the target of the transition is composite. In this case the transition can lead to any descendant state, whereas the semantics of statecharts specifies that the transition should lead to the initial states of each region. This problem can be easily solved by replacing X bits with 0 bits in $enc(trgt(t))$, assuming that initial states are numbered with 0.

After defining *form* for transitions, it can be defined for the whole transition relation: $form(Tr)_i = \bigvee_{t \in Tr} form(t)_i$. Furthermore, for a statechart Sc with the initial configuration c_I, the formula for k-reachability $form(Sc,k)$ can be defined as $form(c_I) \wedge \left(\bigwedge_{i=0}^{k} form(Tr)_i\right)$. Note, that this formula is also suitable for state space exploration by replacing c_I with the actual configuration and setting $k = 1$.

4 Applying CEGAR to Hierarchical Statecharts

The techniques presented in Section 3 can check statecharts against reachability requirements. However, the efficiency (or even termination) of those algorithms is not guaranteed for statecharts with huge or infinite state space. In this chapter we propose an adaption of the Counterexample-Guided Abstraction Refinement (CEGAR) method (Section 2.3) that can be applied to hierarchical statecharts.

4.1 Abstraction of Statecharts

In order to apply CEGAR to statecharts, an over-approximating abstraction [9] has to be defined first. The top-down design of systems involves an intuitive abstraction by first defining top level components and then expanding their inner behavior. For statecharts, this top-down design results in hierarchy, which provides a natural and intuitive abstraction possibility: we can obtain abstract statecharts by only expanding composite states up to a certain depth in the hierarchy. Our encoding described in the previous section supports this idea: different levels are encoded with disjoint sets of variables and formulas, therefore some of these sets do not need to be considered when the corresponding hierarchy level is not expanded.

Formally, we introduce a *state abstraction function* $\mathbf{h}_S : S \mapsto S$ such that $\mathbf{h}_S(s) \in \{s\} \cup anc(s)$, i.e., it maps each state to an *abstract* state, which is either s itself or one of its ancestors. A state s is called *abstracted* if $\mathbf{h}_S(s) \in anc(s)$ and *refined* if $\mathbf{h}_S(s) = s$. If a composite state s is abstracted then $\mathbf{h}_S(s') = \mathbf{h}_S(s)$ must hold for all $s' \in desc(s)$, i.e., descendant states of a composite abstracted state are also abstracted and mapped to the same abstract state as their ancestor.

Besides abstracting states, we also apply abstraction to the variables based on the idea of Clarke et al. [11], originally described for transitions systems. We adapted this approach to statecharts by defining abstraction and refinement methods. In our case abstraction means that variables V of the statechart are partitioned into two disjoint sets: *visible* and *invisible* variables. In the abstract statechart, only visible variables are considered. Formally, a *variable abstraction function* $\mathbf{h}_V : V \mapsto \{\top, \bot\}$ is defined that assigns $\top$ to visible variables and $\bot$ to invisible variables.

The state and variable abstraction functions can be combined into a single *abstraction function* $\mathbf{h} = \{\mathbf{h}_S, \mathbf{h}_V\}$, where $\mathbf{h}(s) = \mathbf{h}_S(s)$ for a state $s \in S$ and $\mathbf{h}(v) = \mathbf{h}_V(v)$ for a variable $v \in V$.

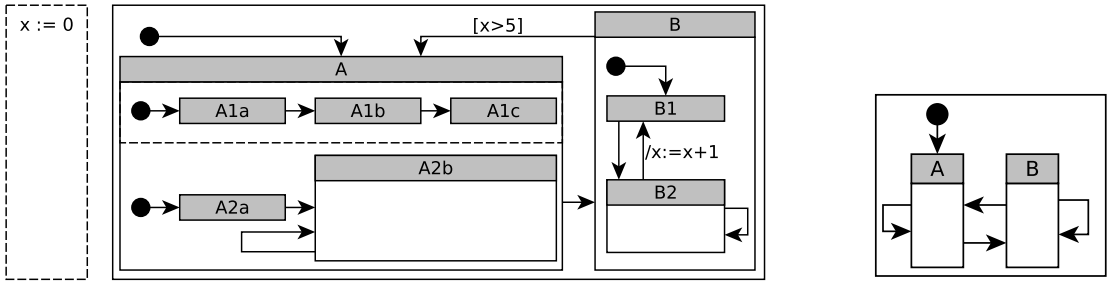
Let $Sc = (S, R, par, I, V, Tr)$ be a statechart and let $\mathbf{h}$ be an abstraction function over Sc. The *abstract statechart* $\mathbf{h}(Sc) = \hat{Sc} = (\hat{S}, \hat{R}, \hat{par}, \hat{I}, \hat{V}, \hat{Tr})$ of Sc corresponding to $\mathbf{h}$ is defined in the following way.

- $\hat{S} = \{\mathbf{h}(s) \mid s \in S\}$, i.e., only the abstract states are kept,
- $\hat{R} = \{r \mid \exists s \in chld(r)$ such that $\mathbf{h}(s) = s\}$, i.e., only regions containing refined states are kept,
- $\hat{par} = par \cap (\hat{S} \times \hat{R} \cup \hat{R} \times \hat{S})$, i.e., hierarchy is preserved between states and regions,

- $\hat{I} : \hat{S} \mapsto \hat{R}$ such that $\hat{I}(\hat{r}) = I(\hat{r})$ for each $\hat{r} \in \hat{R}$, i.e., the initial states of kept regions remain the same,
- $\hat{V} = \{v \in V \mid \mathbf{h}(v) = \top\}$, i.e., only visible variables are kept,
- $\hat{Tr} = \{(\mathbf{h}(src(t)), \mathbf{h}(trgt(t)), trig(t), grd(t), Act(t)) \mid t \in Tr\}$, i.e., sources and targets of transitions are mapped to abstract states. In guard and action expressions, each occurrence of an invisible variable is replaced with a unique constant, so constraints they represent are released.

For a set of states ω, abstraction is defined as $\mathbf{h}(\omega) = \{\mathbf{h}(s) \mid s \in \omega\}$, i.e., the set of abstract states. For a configuration $c = (\omega, \rho, \alpha)$ of the statechart $Sc = (S, R, par, I, V, Tr)$, abstraction is defined as $\mathbf{h}(c) = (\mathbf{h}(\omega), \rho, \{\alpha(v) \mid v \in \hat{V}\})$, i.e, the set of active states is abstracted, the active events are kept, and only visible variables are kept. Note that $\mathbf{h}(c)$ is a configuration for $\mathbf{h}(Sc)$ if c is a configuration for Sc. Furthermore, for a set of configurations C_{Sc} let $\mathbf{h}(C_{Sc}) = \{\mathbf{h}(c) \mid c \in C_{Sc}\}$, i.e., the set of abstracted configurations. For a path $\pi = (c_0, c_1, \ldots, c_n)$ let the *abstract path* $\mathbf{h}(\pi)$ be $\hat{\pi} = (\mathbf{h}(c_0), \mathbf{h}(c_1), \ldots, \mathbf{h}(c_n))$, i.e., the sequence of abstract states.

Recall the example statechart presented in Figure 1. Two possible abstractions for this example can be seen in Figure 3. The abstraction show in Figure 3a is a finer one, with all the variables visible and states refined, except for states in *A2b* and *B2*. In contrast, the statechart in Figure 3b corresponds to a coarser abstraction, with all variables hidden, and only the top level states *A* and *B* being refined.



(a) A finer abstraction.

(b) A coarser abstraction.

Figure 3: Two possible abstract statecharts for the example presented in Figure 1.

Using the abstractions defined above, we can apply the CEGAR approach to statecharts. The following sections describe the main steps of the algorithm specialized for statecharts.

4.2 Initial Abstraction

The input of the algorithm is a statechart $Sc = (S, R, par, I, V, Tr)$ and a set of error configurations C_f. This set can be explicitly given by enumerating error configurations, but it can also be defined by only bounding a subset of states and variables (e.g., $x = 1$ marks all configurations where the value of x is 1). The first step of the algorithm is to create an initial abstraction $\mathbf{h}_0 = \{\mathbf{h}_{S0}, \mathbf{h}_{V0}\}$. In our work, we defined two kind of abstractions: *states-only abstraction* that only abstracts states and *generic abstraction* that abstracts both states and variables. Note, that the former approach can be considered as a special case for the latter one with all variables being visible ($\mathbf{h}_V \equiv \top$).

CEGAR-based algorithms usually start from a coarse abstraction in order to avoid complexity. For statecharts, we achieve this by abstracting each state, except states in the top-level regions. Formally, $\mathbf{h}_{S0}(s) = s$ if $depth(s) = 1$ and $\mathbf{h}_{S0}(s) = \mathbf{h}_{S0}(par(s))$ otherwise.

The initial abstraction for the two types of abstractions is different for $\mathbf{h}_V$, as for states-only abstraction $\mathbf{h}_V \equiv \top$, i.e., all variables are visible. In case of the generic abstraction, only those variables are visible that are bounded when defining the set of error configurations C_f.

4.3 Model Checking

The input of the model checking step is the abstracted statechart $\hat{Sc}$, and the set of abstract error configurations $\hat{C}_f$. The verification of the model can be either performed by exploring the abstract state space or by bounded model checking as described in Section 2.2.[1] Due to the existential property of the abstraction, each concrete path in the statechart has its corresponding abstract path by simply mapping the states and transitions to their abstractions. Therefore, if no error configuration can be reached in the abstract statechart $\hat{Sc}$, then it cannot be reached in the concrete statechart and the algorithm reports that the statechart is safe. On the other hand, if an error configuration in $\hat{c} \in \hat{C}_f$ can be reached, an abstract path $\hat{\pi}_{\hat{c}}$ leading to $\hat{c}$ is returned as a counterexample. However, it is not guaranteed that there is a corresponding concrete path $\pi = (c_0, c_1, \ldots, c_n)$ in Sc such that $\mathbf{h}(\pi) = \hat{\pi}_{\hat{c}}$.

4.4 Concretizing the Counterexample

If model checking marks an error configuration reachable, and provides an abstract counterexample $\hat{\pi} = (\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_n)$, it has to be verified if a corresponding path exists in the original statechart Sc.

We do this by iteratively checking the existence of an $0 \leq i \leq n$ long path $\pi_i = (c_0, c_1, \ldots, c_i)$ in Sc, such that $\mathbf{h}(c_j) = \hat{c}_j$ for $0 \leq j \leq i$. As the length of the searched path is bounded by i, bounded model checking can be applied here. However, the search of bounded model checking can be narrowed in this case, since we know the abstract configurations through which the concrete path must pass. Therefore, besides unfolding the transition relation, the encoding of the abstract states, events and the variable assignment formulas are also joined for each abstract configuration.

If concretization succeeds, the concretized counterexample is returned. Otherwise, if concretization succeeds until the ith iteration, but fails in the $(i+1)$th iteration, then $\hat{c}_i$ is called a *failure configuration* and it is returned as a witness for the counterexample being spurious.

4.5 Refinement

If the counterexample is spurious, the abstraction has to be refined based on the failure configuration $\hat{c} = (\hat{\omega}, \hat{\rho}, \hat{\alpha})$. The abstraction function $\mathbf{h} = \{\mathbf{h}_S, \mathbf{h}_V\}$ consists of two components, which can both be refined.

The state abstraction function $\mathbf{h}_S$ is refined by expanding one more level of the hierarchy in states that are included in the failure configuration, i.e., refining their direct descendants. Formally, the refined state abstraction function $\mathbf{h}'_S$ is defined as $\mathbf{h}'_S(s) = s$ if $par(par(s)) \in \hat{\omega}$ and $\mathbf{h}'_S(s) = \mathbf{h}_S(s)$ otherwise. For the states-only abstraction only this refinement technique can be used as it contains no invisible variables. Consider the example presented in Figure 4a. If the state s_1 is not refined, the state s_2 appears in reachable configurations as there is a path to it through s_1, however the concretization of abstract paths through s_1 will fail, as s_{1b} is not reachable and s_2 is only reachable from that state.

[1] During bounded model checking, a limit for k has to be determined, otherwise the algorithm will not terminate if the transition relation contains a cycle and the error configuration is not reachable. An upper bound for k can possibly be determined by the diameter of the system [5].

(a) Failure due to state abstraction.

(b) Failure due to variable abstraction.

Figure 4: Examples for failed concretization.

For the generic abstraction $\mathbf{h}_V$ can be refined as well, however we only refine $\mathbf{h}_V$ if $\mathbf{h}_S$ cannot be refined anymore, i.e., we prefer to refine the hierarchy first. It can be seen that if a configuration is a failure configuration, and all the active states in the configuration are completely refined, but the execution cannot continue to the next abstract configuration, it is due to guard expressions that contain invisible variables. In this case, we refine variables that appear in guards $grd(t)$ on transitions t with $src(t) \in \hat{\omega}$, i.e., outgoing transitions from states in the error configuration.

Consider the example presented in Figure 4b. If the variable x is invisible, and from state s_1 there is a transition to s_2 with the guard $x = 0$, but the state s_1 only appears in reachable configurations with $x = 1$, then the transition cannot fire. However, if x is abstracted, there is a transition from configuration s_1 to s_2, because there is a transition from $(s_1, x = 0)$ to $(s_2, x = 0)$.

After the refinement, execution continues with the next iteration. The CEGAR loop must eventually terminate since each refinement step either refines a state or a variable, and there are finite states and variables in the statechart. However, the model checking phase may not terminate if the abstract state space is infinite, which can happen if there is an unbounded visible variable.

5 Evaluation

A prototype of the algorithms described in Section 3 and 4 has been implemented in Java using Z3 [20] as the underlying SMT solver. We did not compare our implementation to other tools as our current goal was to demonstrate and compare the applicability of the algorithms presented in the paper. The implementation consists of two different abstraction-refinement pairs, the states-only abstraction, abbreviated in diagrams as *STT* and the generic abstraction referred to as *GEN*. For model checking, four different algorithms have been implemented (summarized in Table 1).

Table 1: Summary of the model checking algorithms.

Name	**Abbreviation**	**Description**
Many-at-once (non-popping)	MON	A naive implementation of state space exploration, that explores all reachable configurations with an SMT solver.
Many-at-once (popping)	MOP	A state space explorer implementation, that uses the push-pop functionality of the solver to efficiently construct the transition relation formula.
One-at-once	OAO	An optimized implementation of state space exploration, that only explores one reachable configuration when turning to the solver.
Bounded model checker	BMC	An implementation that uses bounded model checking.

We tested the performance of the implemented algorithms by checking reachability queries on a statechart that represents a part of the industrial control system described in [2, 21]. This system rep-

resents the safety logic of a power plant, originally described as a functional block diagram. We chose this example as it contains a wide variety of the currently supported elements of statecharts: 3 hierarchy levels with a total of 27 states (5 composite, 22 simple), 9 regions (with the maximum number of parallel regions being 4), 16 variables (13 Boolean, 3 integer) and 27 transitions. Although the structure of the statechart is not so large, the parallelism and the variables induce a large number of configurations.

The metrics measured during each verification run are the time elapsed until termination (*Time*), the number of CEGAR iterations (*Iter*), the maximum number of explored configurations in any iteration (*Confs(max)*) and the number of explored configurations at the last iteration (*Confs(eve)*). Note, that the latter two metrics are not applicable for bounded model checking. An alternative metric could be the length of the path, however it only refers to the depth of the search, not the breadth.

The controller contains signal holders represented with counters, whose maximum value can be parameterized. By adjusting this parameter, the size of the state space can be varied. Measurements with different parameter values have been carried out. Table 2 summarizes the results with parameter value 2.

Table 2: Results for parameter value 2.

Checker	**Abstraction**	**Time (s)**	**Iter**	**Confs(max)**	**Confs(eve)**
MON	STT	timeout	2	8610	8610
MOP	STT	1398.63	5	17036	2855
OAO	STT	1250.226	5	17036	2855
BMC	STT	211.499	5	-	-
MON	GEN	48.389	12	1484	1484
MOP	GEN	37.817	12	1484	1484
OAO	GEN	8.942	12	1484	1484
BMC	GEN	77.478	12	-	-

The table shows that the one-at-once state space explorer, which is based on the optimized state space exploration algorithm outperforms the other two exploration methods. Amongst the other two exploration implementations, the popping version performs slightly better than the non-popping one, which even fails to terminate with states-only abstraction. In case of the abstraction methods, the generic abstraction has better results regarding every metric with every model checker, justifying the usefulness of combining variable abstraction with state abstraction. Note that with the generic abstraction, the bounded model checker is the least effective, however with all the variables visible, it performs the best. The improvement is relative though, as it is still approximately three times slower than with the generic abstraction. The reason behind this is that the solver can perform more efficient search in the state space than the exploring algorithms.

In order to examine the scalability of the algorithms, they have been ran with different values of the parameter. The comparison of the execution times for the different model checking methods with generic abstraction can be seen in Figure 5.

It turns out that the state space exploration based algorithms perform better than BMC. Amongst those three, the many-at-once implementation, that does not use the push-pop functionality of the solver is the least effective, as it is remarkably slower for every parameter value than the other two, and fails to terminate for every value greater than 10. For small parameter values, the one-at-once explorer performs better, however for bigger parameter values, the many-at-once implementation, that uses the solver's pop-push functionality performs better.

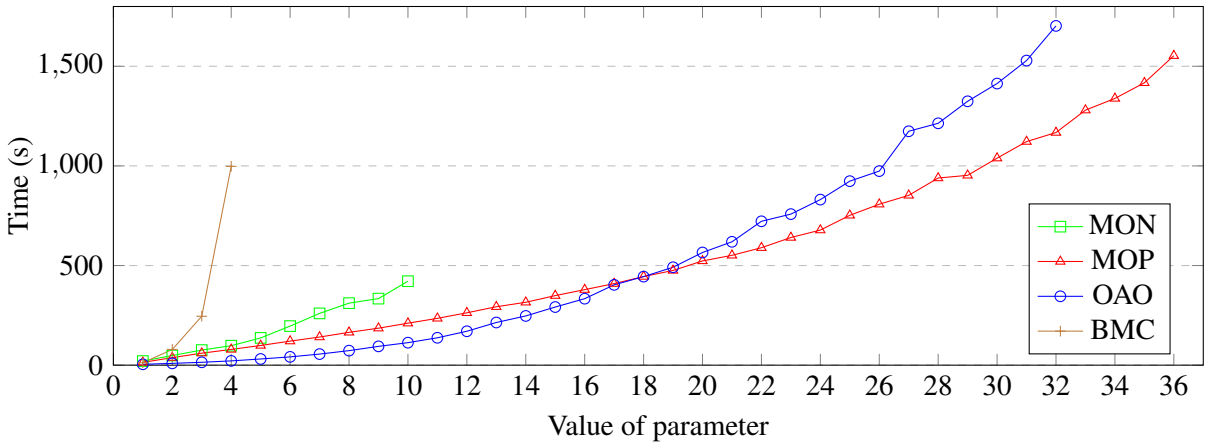


Figure 5: Comparison of execution time for different model checking methods.

6 Conclusions

In our paper we proposed a novel adaptation of the Counterexample-Guided Abstraction Refinement (CEGAR) algorithm applied to the reachability analysis of hierarchical statecharts. From the theoretical point of view, we proposed an encoding of statecharts into logical formulas that preserves information about the hierarchy of states. This encoding was effectively used in implementing abstraction and refinement techniques that utilize the hierarchical structure of statecharts. We showed that this approach allows us to use SMT solvers to check the abstract model, thus to apply full state space exploration and also bounded model checking. Furthermore, we also combined this method with the abstraction and refinement of variables in the model. On the practical side, we implemented and evaluated our new algorithms on an industry-motivated example, demonstrating their applicability.

Although the algorithms proved to be applicable, there are several opportunities for improvement. The set of the supported statechart elements can be extended with the history indicator and allowing more sophisticated event queue models. Further abstractions can be introduced, for example predicate abstraction [12] over variables in the statechart. The refinement methods could also be extended, for example with interpolation [17] or unsat core-based variable refinement [16]. It would also be beneficial to compare our algorithms to different approaches to see the advantages or drawbacks of CEGAR and SMT-based model checking.

References

[1] Rajeev Alur, Michael McDougall & Zijiang Yang (2002): *Exploiting Behavioral Hierarchy for Efficient Model Checking*, pp. 338–342. Lecture Notes in Computer Science, Springer, doi:10.1007/3-540-45657-0_-25.

[2] Tamás Bartha, András Vörös, Attila Jámbor & Dániel Darvas (2012): *Verification of an Industrial Safety Function Using Coloured Petri Nets and Model Checking*. In: *Proceedings of the 14th International Conference on Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2012)*, Hungarian Academy of Sciences, Computer and Automation Research Institute, pp. 472–485.

[3] Dirk Beyer & Stefan Löwe (2013): *Explicit-State Software Model Checking Based on CEGAR and Interpolation*. In: *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science* 7793, Springer, pp. 146–162, doi:10.1007/978-3-642-37057-1_11.

[4] Purandar Bhaduri & S. Ramesh (2004): *Model Checking of Statechart Models: Survey and Research Directions*. *CoRR* cs.SE/0407038. Available at `http://arxiv.org/abs/cs.SE/0407038`.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke & Yunshan Zhu (1999): *Symbolic Model Checking without BDDs*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 1579, Springer, pp. 193–207, doi:10.1007/3-540-49059-0_14.

[6] Aaron R Bradley & Zohar Manna (2007): *The calculus of computation: Decision procedures with applications to verification*. Springer, doi:10.1007/978-3-540-74113-8.

[7] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin & J. D. Reese (1998): *Model checking large software specifications*. *IEEE Transactions on Software Engineering* 24(7), pp. 498–520, doi:10.1109/32.708566.

[8] Edmund M Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking*. *Journal of the ACM* 50(5), pp. 752–794, doi:10.1145/876638.876643.

[9] Edmund M Clarke, Orna Grumberg & David E Long (1994): *Model checking and abstraction*. *ACM Transactions on Programming Languages and Systems* 16(5), pp. 1512–1542, doi:10.1145/186025.186051.

[10] Edmund M Clarke, Orna Grumberg & Doron Peled (1999): *Model checking*. MIT Press.

[11] Edmund M Clarke, Anubhav Gupta & Ofer Strichman (2004): *SAT-based counterexample-guided abstraction refinement*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23(7), pp. 1113–1123, doi:10.1109/TCAD.2004.829807.

[12] Susanne Graf & Hassen Saidi (1997): *Construction of abstract state graphs with PVS*. In: *Computer Aided Verification*, *Lecture Notes in Computer Science* 1254, Springer, pp. 72–83, doi:10.1007/3-540-63166-6_10.

[13] Ákos Hajdu, András Vörös & Tamás Bartha (2015): *New search strategies for the Petri net CEGAR approach*. In: *Application and Theory of Petri Nets and Concurrency*, *Lecture Notes in Computer Science* 9115, Springer, pp. 309–328, doi:10.1007/978-3-319-19488-2_16.

[14] Steffen Helke & Florian Kammüller (2016): *Verification of statecharts using data abstraction*. *International Journal of Advanced Computer Science and Applications* 7(1), pp. 571–583, doi:10.14569/IJACSA.2016.070179.

[15] Diego Latella, Istvan Majzik & Mieke Massink (1999): *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker*. *Formal Aspects of Computing* 11(6), pp. 637–664, doi:10.1007/s001659970003.

[16] Martin Leucker, Grigory Markin & MartinR. Neuhäußer (2015): *A New Refinement Strategy for CEGAR-Based Industrial Model Checking*. In: *Hardware and Software: Verification and Testing*, *Lecture Notes in Computer Science* 9434, Springer, pp. 155–170, doi:10.1007/978-3-319-26287-1_10.

[17] K.L. McMillan (2005): *Applications of Craig Interpolants in Model Checking*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 3440, Springer, pp. 1–12, doi:10.1007/11494744_2.

[18] Yael Meller (2016): *Model Checking Techniques for Behavioral UML Models*. Ph.D. thesis, Israel Institute of Technology.

[19] Yael Meller, Orna Grumberg & Karen Yorav (2014): *Verifying Behavioral UML Systems via CEGAR*. In: *Integrated Formal Methods*, Lecture Notes in Computer Science, Springer, pp. 139–154, doi:10.1007/978-3-319-10181-1_9.

[20] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[21] Erzsébet Németh, Tamás Bartha, Csaba Fazekas & Katalin M. Hangos (2009): *Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured Petri nets*. *Reliability Engineering & System Safety* 94(5), pp. 942–953, doi:10.1016/j.ress.2008.10.012.